Breadth First Search on Cost-efficient Multi-GPU Systems

Takuji Mitsuishi
Keio University
3-14-1 Hiyoshi, Yokohama,
223-8522, Japan
mits@am.ics.keio.ac.jp

Jun Suzuki
NEC Corporation
1753, Shimonumabe,
Nakahara-ku, Kawasaki,
Kanagawa 211-8666, Japan
j-suzuki@ax.jp.nec.com

Yuki Hayashi
NEC Corporation
1753, Shimonumabe,
Nakahara-ku, Kawasaki,
Kanagawa 211-8666, Japan
y-hayashi@kv.jp.nec.com

Masaki Kan
NEC Corporation
1753, Shimonumabe,
Nakahara-ku, Kawasaki,
Kanagawa 211-8666, Japan
kan@bq.jp.nec.com

Hideharu Amano
Keio University
3-14-1 Hiyoshi, Yokohama,
223-8522, Japan
asap@am.ics.keio.ac.jp

ABSTRACT

A parallel Breadth First Search (BFS) algorithm is proposed for cost-efficient multi-GPU systems without enough memory amount or communication performance. By using an improved data structure for the duplication elimination of local nodes, both required memory amount and processing time are reduced. By using Unified Virtual Addressing, time for communication can be hidden with the computation. The proposed algorithm is implemented on two cost-efficient multi-GPU systems: Express multi-GPU system which has a full of flexibility but the communication latency between GPU and host is limited, and a high-end gaming machine whose memory is limited. Both systems achieve good strong scaling with the proposed methods. On Express multi-GPU system, the communication overhead was almost completely hidden, and the aggregate communication throughput reached 4.77 GB/sec (38.16 Gbps), almost theoretical maximum.

Keywords

GPU, ExpEther, Breadth First Search, Scalability

1. INTRODUCTION

Big data processing has been applied in various fields from social networks to biology. In such fields, the relationship between data is often represented by large target graphs, that require a large computation cost to analyze. As a cost-efficient solution, multi-GPU systems which have been widely utilized in scientific computing have been researched recently. Graph500, a benchmark with a graph analysis, was adopted as a target application program for ranking supercomputers.

Despite the high performance of supercomputers in which nodes providing a number of GPUs are connected with powerful interconnection networks like Infiniband, they are not suitable for data-centers that work mainly for non-numeric computing considering their cost. Cost-effective alternative is introducing a small scale multi-GPU systems depending on the performance demand. We proposed a Breadth First Search (BFS) algorithm for such a cost-effective multi-GPU system called GPU-BOX [6]. However, GPU-BOX which requires a large cabinet for concentrating a large number of GPUs is still an expensive solution.

Here, we picked up two cost-efficient solutions. One is Express multi-GPU system based on ExpEther [9], a virtualization technique for extending PCI Express. Express multi-GPU system can be built just pushing GPUs into an extension rack of ExpEther. Each rack can hold two GPUs, and by connecting them with ExpEther, the programmers can treat all GPUs as if they were connected with a PCI Express bus of a single host node.

Another target is a high-end gaming machine just connecting four GPUs into PCI Express Gen3 bus of a host. Although the number of GPUs is limited into four in this approach, it is the most cost-efficient approach to get a powerful multi-GPU system.

Both Express multi-GPU system and a high-end gaming machine have a limitation on the memory amount or interconnection network. However, traditional work for developing parallel BFS algorithms for GPUs mainly focuses on a single GPU or a large scale GPU clusters with powerful interconnection network. In this paper, we proposed a parallel BFS algorithm for such cost-efficient multi-GPU systems with memory and network limitation.

Here we introduced the following two methods: (1) For eliminating duplication of neighboring vertices, combination of queue-type and vector-type array is used instead of traditional queue-based algorithms, and we can save memory usage also. (2) The communication is overlapped with the computation by using Unified Virtual Addressing and Circular Left-Right approach. By combination of them, we show that a high performance can be obtained even with cost efficient multi-GPU systems through the evaluation.

The rest of paper is organized as follows: First, cost-efficient multi-GPU systems are introduced in Section 2 as an accelerators in a data-center. Then, the Breath First Search is introduced in Section 3, focusing on parallel processing of a level-synchronized BFS. After a related two different scale BFS algorithms are explained in Section 4, we

This work was presented in part at the international symposium on Highly-Efficient Accelerators and Reconfigurable Technologies (HEART2015) Boston, MA, USA, June 1-2, 2015.

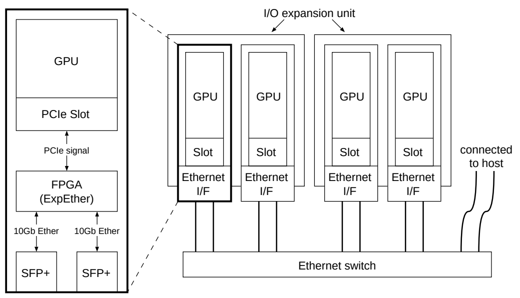


Figure 1: Express multi-GPU system

propose our BFS algorithm in Section 5.

Finally, the evaluation results are shown in Section 6, and Section 7 concludes this paper.

2. COST EFFICIENT MULTI-GPU SYSTEMS

Here, our target multi-GPU systems are introduced.

2.1 Express multi-GPU system

ExpEther [9] is an Ethernet based virtualization technique, which was developed by NEC [7]. It extends PCI Express which is high performance I/O network but limited to small area around the cabinet to much wider area by using Ethernet. Various types of I/O devices on distant location can be connected as if they were inside the cabinet.

As shown in Figure 1, an ExpEther NIC is connected to each PCI Express port of GPU. In the NIC, PEB (PCI Express-to-Ethernet Bridge) is provided. It encapsulates Transaction Layer Packet (TLP) used in PCI Express into Ethernet frame, and decapsulates it for extending PCI Express to Ethernet. Multiple GPUs are connected with a common Ethernet switch. A delay-based protocol which supports repeat and congestion control is utilized instead of TPC, the loss-based protocol. It also employs Go-back-N flow-control instead of Selective-Repeat. By using such a performance centric protocol for small area communication, ExpEther can support larger bandwidth and smaller latency than common Ethernet even using common cables and switches for Ethernet. A host can be connected through ExpEther NIC, and all GPUs can be treated as if they were connected to the PCI Express of the host.

Since ExpEther gives programmer a flat view of multi-GPUs, the programming complexity is much reduced. Programmers do not have to care about the communication between hosts which provide a number of GPUs.

Express multi-GPU system is built by using ExpEther I/O expansion units. The I/O expansion unit provides two slots to ExpEther NIC implemented with an FPGA. For each slot, since two 10Gb Ethernet ports (SFP+) are provided, thus, 20Gbps bandwidth is available in total. Express multi-GPU system enables to treat a lot of GPUs located in the distant places. Here, two I/O expansion units are used to connect four GPUs in total. In Express multi-GPU systems, data can be directly transferred between GPUs thorough ExpEther. Considering the latency of the ExpEther NIC and Ethernet, the latency between GPUs is short [8]. However, the communication latency between host and GPUs is large and it sometimes bottlenecks the system.

2.2 High-end gaming machine

One of the best ways to get the high performance with small amount of money is to use GPUs for gaming machines. Here, the target high-end gaming machine connects four GeForce GPUs with PCIe Gen3 slots on a commodity host CPU. Unlike Express multi-GPU systems, the scale of the system is limited into four. However, the host and four GPUs are connected tightly with PCIe Gen3 and so the communication latency is short. Memory is limited due to the GPUs for gaming machines.

Algorithm 1 Level-synchronized BFS

```
$CF/NF$ is current/next frontier.
$Visited$ indicates whether the vertex has been visited.
$Label$ is searching result.
$src$ is source vertex.

/* Initialize part */
Initialize $Visited$ to 0 and $Label$ to -1
push($NF$, $src$); $Visited[src] \leftarrow 1$; $Label[src] \leftarrow src$
/* BFS iteration */
while $NF$ is not empty do
  $CF \leftarrow NF$; $NF \leftarrow \phi$
  for all $u$ in $CF$ in parallel do
    $u \leftarrow$ pop($CF$)
    for $v$ that is adjacent to $u$ do
      if $Visited[v]$ is zero then
        $Visited[v] \leftarrow 1$
        $Label[v] \leftarrow u$
        push($NF$, $v$)
      end if
    end for
  end for
end while
```

Table 1: Example of level-synchronized BFS: search undirected graph in Fig 2 from vertex 0

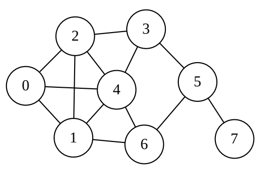
BFS iteration	CF	NF	$Label$
0		0	0,-1,-1,-1,-1,-1,-1,-1
1	0	1, 2, 4	0, 0, 0,-1, 0,-1,-1,-1
2	1, 2, 4	3, 6	0, 0, 0, 2, 0,-1, 1,-1
3	3, 6	5	0, 0, 0, 2, 0, 3, 1,-1
4	5	7	0, 0, 0, 2, 0, 3, 1, 5
5	7		0, 0, 0, 2, 0, 3, 1, 5

3. BREADTH FIRST SEARCH

Breadth first search, BFS is a typical graph algorithm that searches all linked vertices one after another from a source vertex, that is root. Each visited vertex is labeled by its parent identifier or distance from source vertex as searching result. We use parent labeling in this paper since Graph500 benchmark [1] adopts it.

3.1 Level-synchronized BFS

For solving BFS with parallel architecture like GPU, we apply level-synchronized BFS as shown in Algorithm 1. The frontiers CF/NF in Algorithm 1 is boundary between visited and unvisited vertices and plays a role in termination condition of searching. Table 1 shows an example result. We use this example to explain our algorithm in Section 5.



$$A = \begin{bmatrix} 0 & 1 & 1 & 0 & 1 & 0 & 0 & 0 \\ 1 & 0 & 1 & 0 & 1 & 0 & 1 & 0 \\ 1 & 1 & 0 & 1 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 1 & 1 & 0 & 0 \\ 1 & 1 & 1 & 1 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 1 & 1 \\ 0 & 1 & 0 & 0 & 1 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \end{bmatrix}$$

$$C = \{ 1,2,4,0,2,4,6,0,1,3,4,2,4,5,0,1,2,3,6,3,6,7,1,4,5,5 \}$$
$$R = \{ 0,3,7,11,14,19,22,25,26 \}$$

Figure 2: CSR

3.2 CSR

Graphs are represented by an adjacency matrix A which is a sparse matrix. For saving memory usage, we transform an adjacency matrix into Compressed Sparse Row, or CSR. It is a sparse matrix format constructed of two arrays column-indices C and row-offsets R [5]. As shown in Figure 2, C has every adjacency lists successively and R points to start of each adjacency list in C. The length of C is M, and the length of R is $N+1$, where M is the number of edges and N is the number of vertices in the graph.

4. RELATED WORK

Several parallel BFS algorithms have been recently proposed for various scale multi-GPU systems.

For example, Merrill et al. [5] proposed a BFS algorithm for single host systems including four GPUs. They proposed a technique for efficiently gathering neighbors by changing methods in accordance with the task granularity. However, large scale systems with a number of GPUs that cannot be implemented in a machine cabinet are not considered. They do not take into account communication between GPUs for scalability.

In contrast, Ueno et al. [10] proposed a BFS algorithm for TSUBAME2.0 supercomputer that has a large number of (4096) GPUs. Each node of TSUBAME2.0 has three GPUs and two CPUs, and eight CPU threads are assigned into each GPU. The total processing is distributed into CPUs and GPUs. To suppress the communication frequency, 2D partitioning is adopted. Their algorithm is optimized to a large scale cluster with a large number of GPUs and CPUs, but it is not suitable for our target machines with relatively poor communication performance.

5. ALGORITHM AND IMPLEMENTATION

5.1 Overview of algorithm

Algorithm 2 shows a pseudo code of our parallel BFS algorithm for cost-efficient multi-GPU systems. We exchange vertices at GatherNeighbors phase to hide communication as much as possible for Express multi-GPU system since GatherNeighbors occupies the most part of computation time.

Before going into detail, we explain the data structure used in the algorithm. *Label* is an 8-byte array whose indices correspond to the vertex identifier. We call this data structure vector-type array. It holds parent vertex identifiers and returned search results. $Vector$ is an 8-byte vector-type array that holds parent vertex identifiers like *Label* and is initialized to a negative one at every BFS iteration. $Visited$ is a 1-byte vector-type array that represents whether the vertex has been visited; zero for unvisited and one for visited. *Queue* is an 8-byte array that has vertex identifiers in the current frontier. The length of *Queue* is variable, and we call this data structure queue-type array. Each array is allocated {8 or 1}-byte $\times N/p$ bytes on each GPU memory, where N is the number of vertices and p is the number of GPUs.

Algorithm 2 Overview of proposed BFS on each GPU_i

```
/* Initialize part */
Initialize $Label_i$ to -1, $Visited_i$ to 0 on each $GPU_i$
**if** the $GPU_i$ has source vertex **then**
    Enqueue($Queue_i$, $src$)
    $Label_i[src] \leftarrow src$
    $Visited_i[src] \leftarrow 1$
**end if**
/* BFS iteration */
**while** any $Queue_i$ is not empty **do**
    Initialize $Vector_i$ to -1
    // GatherNeighbors
    **for all** $j$ ($0 \le j < NumGPUs$) **do**
        **for all** $u$ in $Queue_j$ in parallel **do**
            $u \leftarrow$ Dequeue($Queue_j$)
            **for all** $offset$ ($R[u] \le offset < R[u+1]$) **do**
                $Vector_i[C[offset]] \leftarrow u$
            **end for**
        **end for**
    **end for**
    // UpdateLabels
    **for all** $id$ that is id for vector-type array in parallel **do**
        **if** $Vector_i[id]$ is not negative value **then**
            **if** $Visited_i[id]$ is one **then**
                $Vector_i[id] \leftarrow -1$
            **end if**
        **else**
            $Label_i[id] \leftarrow Vector_i[id]$
            $Visited_i[id] \leftarrow 1$
        **end if**
    **end for**
    // ConvertVtoQ
    **for all** $id$ that is id for vector-type array in parallel **do**
        **if** $Vector_i[id]$ is not negative value **then**
            Enqueue($Queue_i$, $id$)
        **end if**
    **end for**
**end while**
```

5.2 Cost-efficient GatherNeighbors

Here, we explain implementation of gathering and scattering neighbors, which prunes efficiently duplicate vertices and saves memory usage.

We use Merrill et al.'s gathering method [5] which selects an approach corresponding to the task granularity since [5] is sophisticated for GPU architecture.

However, the naive implementation of the method produces a lot of duplicate vertices. In this case, we have to prune duplications for reducing communication. Merrill et al. do not consider pruning duplications so much since they can use fast PCI Express bus for communication

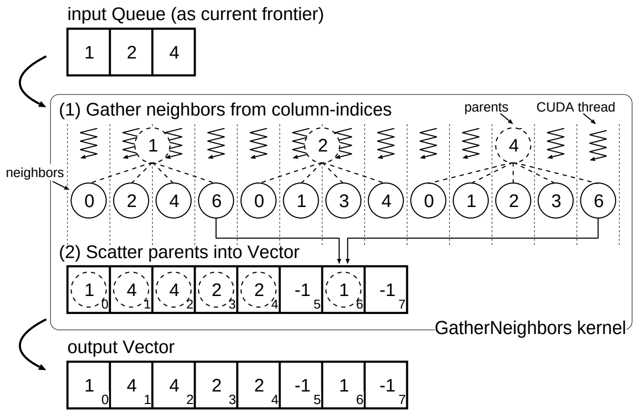

input Queue (as current frontier)
1 2 4
(1) Gather neighbors from column-indices
parents
CUDA thread
neighbors
(2) Scatter parents into Vector
GatherNeighbors kernel
output Vector
1 4 4 2 2 -1 1 -1

Figure 3: GatherNeighbors: This situation is same with iteration 2 at Table 1 on single GPU

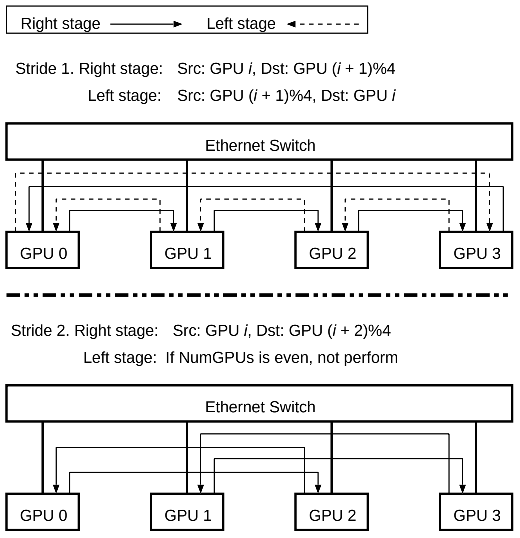

Right stage
Left stage
Stride 1. Right stage: Src: GPU i, Dst: GPU (i + 1)%4
Left stage: Src: GPU (i + 1)%4, Dst: GPU i
Ethernet Switch
GPU 0
GPU 1
GPU 2
GPU 3
Stride 2. Right stage: Src: GPU i, Dst: GPU (i + 2)%4
Left stage: If NumGPUs is even, not perform
Ethernet Switch
GPU 0
GPU 1
GPU 2
GPU 3

Figure 4: Circular Left-Right approach with four GPUs

between GPUs. Mastrostefano and Bernaschi prune duplications completely with a Sort-Unique method [4] to reduce communication overhead between inter node. However, it is a large task that can dominate the total execution time.

Here, we propose scattering parents method for pruning duplications without time-consuming filter processing. Figure 3 shows our scattering method. Parents of gathered neighbors are scattered into vector-type array instead of enqueuing neighbors to the queue. If we gather duplications, parents of them are stored in the same place like Figure 3.

Moreover, global memory usage on each GPU is reduced $(M - N)/p \times 8$ bytes by replacing the queue to vector-type array. For example, M is $edgefactor$ times N at Graph500 benchmark, and default $edgefactor$ is 16. To handle large graphs by the high-end gaming machine without enough memory, work storage size should not depend on M corresponding to several times of N.

5.3 Hiding communication with Circular Left-Right approach

We hide communication efficiently by GatherNeighbors phase using Unified Virtual Addressing (UVA) and Circular Left-Right approach.

Hiding inter node communication has been a common technique used in [10]. On the other hand, intra node communication has not been considered thoroughly, since it does not affect performance seriously. However, we have to care about it because Express multi-GPU systems don't have sufficient communication bandwidth.

UVA enables to use a unified address space by virtually integrating memories of a CPU and GPUs. We can copy data directly between two different GPU memories without copying data in CPU memory temporally, and a GPU can directly access the memory of other GPUs in kernels. We call the former function UVA Memory Copy Peer (UVA-MCP) and the latter UVA Memory Access Peer (UVA-MAP). Using UVA-MCP, we can control communication and computation flexibly, but receiving buffer is required on each GPU. On the other hand, we can implement easily and need not additional buffer with UVA-MAP.

The circular left-right approach is our proposed communication method applied left-right approach [2] for each GPU to communicate with all GPUs efficiently. Our previous work presents improved method of [2] for all-to-all communication. We can achieve large aggregate throughput furthermore since all GPUs send and receive at the same time without contention at each stage. Figure 4 shows the communication between four GPUs. GPU i $(0 \leq i < p)$ communicates with GPU $(i + j)\%p$, where j is a variable for stride $(1 \leq j \leq \lfloor p/2 \rfloor)$. If the number of GPUs p is even, the left stage communication at the last stride is omitted since it is the same.

When we perform the GatherNeighbors phase with UVA-MCP and the circular left-right approach, first, GPU i processes its own $Queue_i$ at the kernel, while it receives peer $Queue_j$ from GPU j by cudaMemcpyPeerAsync. Then GPU i processes $Queue_j$ received just before, receives $Queue_k$, and iterates until each GPU processes all $Queue$ similarly. We create two CUDA streams for the kernel and copy, and synchronize both streams by cudaStreamSynchronize, at the end of each kernel and copy. By UVA-MAP, GPU i accesses $Queue_j$ on GPU j in the kernel in accordance with the circular left-right approach.

In this paper, we only use UVA-MCP because we can not control communication explicitly by using UVA-MAP. On conventional single host multi-GPU system such as gaming machine in the next section, there are almost no performance difference between UVA-MCP and UVA-MAP. On the other hand, the performance of UVA-MCP is better than one of UVA-MAP on Express multi-GPU system.

5.4 Other implementation issues

5.4.1 UpdateLabels

When $Vector[id]$ is not negative and $Visited[id]$ is zero meaning that the vertex is gathered at the BFS iteration but unvisited, the vertex's $Label[id]$ and $Visited[id]$ are updated. If $Vector[id]$ value is not negative and $Visited[id]$ value is one, the vertex's $Vector[id]$ is initialized to a negative one to avoid enqueuing visited vertices to $Queue[id]$ at ConvertVtoQ.

5.4.2 ConvertVtoQ

Valid vertices in $Vector$ are enqueued to $Queue$ as the

Table 2: Evaluation Environment (Express system)

CPU	Intel Xeon E5-1650 @ 3.20GHz
Host Memory	16GB
OS	CentOS 6.3
CUDA	Toolkit 5.5
GPU	NVIDIA Tesla K20 ×4
ExpEther board	NEC N8007-104
I/O expansion unit	NEC N8000-1005 ×2
Switch	Mellanox SX1012 ×2
Network	10Gb Ethernet ×2

Table 3: Evaluation Environment (Gaming)

CPU	Intel Core i7-4770K @ 3.50GHz
Host Memory	16GB
OS	Ubuntu 14.04
CUDA	Toolkit 5.5
GPU	NVIDIA GeForce GTX 660 ×4
Motherboard	ASUS Maximus VI Extream
PCI Express	Generation 3.0
SLI adapter	Not use

Table 4: TEPS with Merrill's BFS (SCALE=20, edgefactor=96)

	Merrill		Gaming		Express	
GPUs	GTEPS	Speedup	GTEPS	Speedup	GTEPS	Speedup
1	3.1	-	0.14	-	0.32	-
2	4.4	1.4	0.81	5.7	1.13	3.5
4	6.2	1.4	1.62	2.0	1.99	1.8

next frontier. We use scan operation to calculate offsets for enqueuing to *Queue*. [6] describes this method in detail.

6. EVALUATION

6.1 Evaluation environment

We evaluated our proposed parallel BFS algorithm on two systems. Both systems have a single CPU and four homogeneous GPUs. One is the Express multi-GPU system shown in Table 2. It consists of two NEC N8000-1005 expansion units and two switches. Each GPU can use bidirectional 10 Gbps (1.25 GB/sec) transfer capability. The other is a high-end gaming machine shown in Table 3. All GPUs are inserted into PCI Express Gen3 slots of the host motherboard.

Graph500 benchmark is used as target graphs in the evaluation. It generates Kronecker graph [3] decided by parameters including $SCALE$ and $edgefactor$ ($N = 2^{SCALE}$, $M = N \times edgefactor$). We used default parameters without $SCALE$ and $edgefactor$ in evaluations. Here, searched graph data (CSR) were stored into GPU memories before searching.

6.2 Achieved TEPS

Here, we show Traversed Edges Per Second (TEPS), a performance measure of graph search algorithms, in the execution of two systems with different numbers of GPUs.

Figure 6 and Figure 7 show TEPS with $edgefactor = 16$ of the gaming machine and the Express multi-GPU system respectively. Those figures show TEPSs are improved as graph size and the number of GPUs grow larger on both systems.

We can bring near the performance on Express multi-GPU system to one on gaming machine because about 85 % and 80 % of communication time are hided by computation time with two GPUs and four GPUs respectively.

Each vertical line in both figures shows strong scaling at each $SCALE$. At $SCALE = 22$ line, TEPS of gaming machine is almost proportional to the number of GPUs. While, TEPS of Express multi-GPU system is not proportional between two and four GPUs. The reason why is the device-to-host transfer of output result, that is *Label*. Figure 8 shows *Label* transfer times and search times that do not include *Label* transfer times at searching graph of $SCALE = 22, edgefactor = 16$ on both systems with some GPUs. On both systems, search times decrease as the number of GPUs grows. However, *Label* transfer times are constant since we can not transfer data from devices to a host in parallel. As shown in Figure 5, the constant time of Express multi-GPU system becomes longer than one of gaming machine. Therefore, the strong scaling of Express multi-GPU system is inferior to one of gaming machine.

6.3 Peer-to-peer communication

Here, we evaluated maximum aggregate throughput of both systems. Figure 10 shows maximum throughput on the Express multi-GPU system with four GPUs at the GatherNeighbors phase when $SCALE = 24$ in the fourth BFS iteration. We can achieve 4.77 GB/sec (38.16 Gbps) in this situation (each GPU sends about 12.9 MB data) on the Express multi-GPU system. On gaming machine, we can achieve 24.04 GB/sec (192.32 Gbps) in the same situation. According to the evaluation of the theoretical maximum shown in [8], the aggregate throughput on both systems are almost at the theoretical maximum.

6.4 Comparison with related work

Here, we compare the proposed BFS with our previous work [6] and one by Merrill et. al. [5].

Our previous BFS improved the BFS by Mastrostefano et. al. [4] on the data transfer size. Figure 9 shows average CUDA kernel execution times with the previous and proposed BFS on gaming machine with a GPU. We can see kernel times of our proposed BFS is about five times faster than those of the previous BFS because of eliminating sort processing. Moreover, the proposed BFS is superior to the previous one on data communication. While the previous BFS can reduce 30 % transfer size than Mastrostefano's one, the proposed BFS can hide 80 % of communication by the computation.

Table 4 shows TEPS values by Merrill's and the proposed BFS with $SCALE = 20, edgefactor = 96$. This shows TEPS of Merrill's BFS is about 3.8 and 3.1 times as the proposed BFS with four GPUs on gaming machine and Express system. However, strong scaling of the proposed BFS on both systems is better than that by Merrill's BFS. When the system size grows, the proposed BFS has a possibility to overcome the Merill's one by using a large number of GPUs.

Most of kernel time of proposed BFS is the time of scattering neighbors into $Vector$ at GatherNeighbors, that CUDA threads write data to global memory at random. The random accesses cause difference of performance between Merrill's BFS and our proposed BFS. However, the number of times of random accesses can be divided by GPUs. The graphs (adjacency matrices) generated by Graph500 have

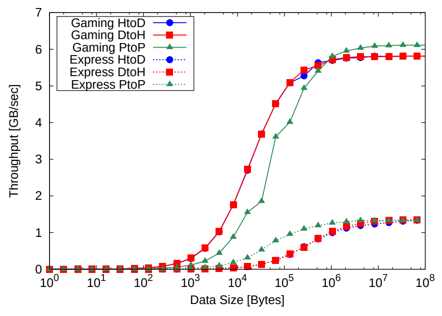


Figure 5: Throughput

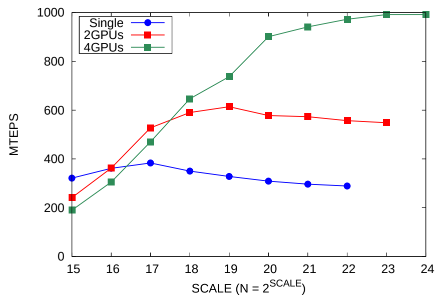


Figure 6: TEPS on Gaming machine

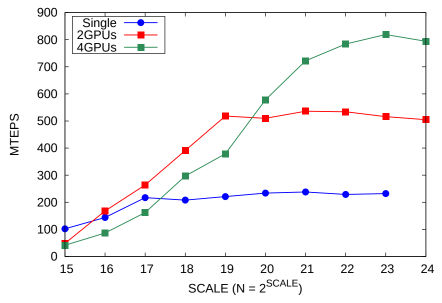


Figure 7: TEPS on Express

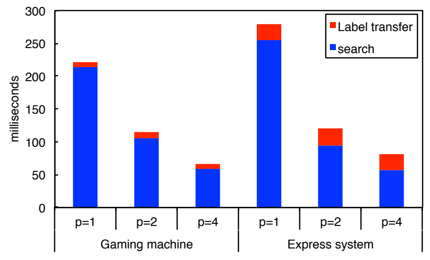


Figure 8: Profiling at $SCALE = 22, edgefactor = 16$

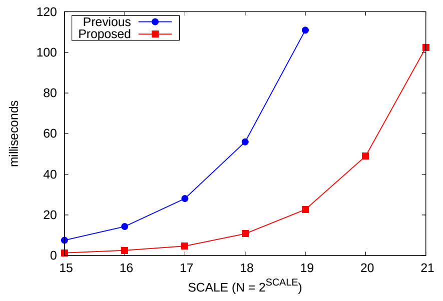


Figure 9: Kernel times

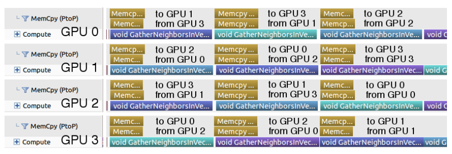


Figure 10: Overlapping communication and computation with four GPUs on Express system

vertices that are relatively evenly spread [5]. Accordingly, each GPU performs almost the same times of random access. Moreover, the more system has GPUs, the range of global memory that can be accessed is the smaller. Thus, the spatial locality will be improved. From the above, we can achieve good strong scaling with the proposed BFS.

7. CONCLUSION

We implemented and evaluated a parallel BFS algorithm on two economical multi-GPU systems, the Express multi-GPU system and the high-end gaming machine. We saved memory usage by combining queue-type and vector-type array. For hiding communication, we used UVA and circular left-right approach to improve aggregate throughput. We can hide about 85 % and 80 % of communication time by computation with two and four GPUs respectively on Express multi-GPU system. Moreover we can achieve good strong scaling with our proposed gathering neighbors method on both systems. Techniques for saving memory and hiding the communication overhead will be useful for other cost-efficient multi-GPU systems which can be easily introduced in data centers.

Our future work is solving bottleneck of transferring computation results to bring performance of the Express multi-GPU system to that of the gaming machine.

8. REFERENCES

[1] Graph 500. http://www.graph500.org/.

[2] L. Barnes. Multi-gpu programming, 2013. http://on-demand.gputechconf.com/gtc/2013/presentations/S3465-Multi-GPU-Programming.pdf.

[3] J. Leskovec, D. Chakrabarti, J. Kleinberg, and C. Faloutsos. Realistic, mathematically tractable graph generation and evolution, using kronecker multiplication. In *PKDD*, pages 133–145. Springer, 2005.

[4] E. Mastrostefano and M. Bernaschi. Efficient breadth first search on multi-gpu systems. *Journal of Parallel and Distributed Computing*, 73(9):1292 – 1305, 2013.

[5] D. Merrill, M. Garland, and A. Grimshaw. Scalable gpu graph traversal. In *Proceedings of the 17th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '12, pages 117–128, New York, NY, USA, 2012. ACM.

[6] T. Mitsuishi, S. Nomura, J. Suzuki, Y. Hayashi, M. Kan, and H. Amano. Accelerating breadth first search on gpu-box. In *International symposium on Highly Efficient Accelerators and Reconfigurable Technologies*, HEART'14, July 2014.

[7] NEC Corporation. http://www.nec.co.jp.

[8] S. Nomura, T. Mitsuishi, J. Suzuki, Y. Hayashi, M. Kan, and H. Amano. Performance analysis of the multi-gpu system with expether. In *International symposium on Highly Efficient Accelerators and Reconfigurable Technologies*, HEART'14, July 2014.

[9] J. Suzuki, Y. Hidaka, J. Higuchi, T. Yoshikawa, and A. Iwata. Expressether - ethernet-based virtualization technology for reconfigurable hardware platform. In *Proceedings of the 14th IEEE Symposium on High-Performance Interconnects*, HOTI '06, pages 45–51, Washington, DC, USA, 2006. IEEE Computer Society.

[10] K. Ueno and T. Suzumura. Parallel distributed breadth first search on gpu. In *High Performance Computing (HiPC), 2013 20th International Conference on*, pages 314–323, Dec 2013.